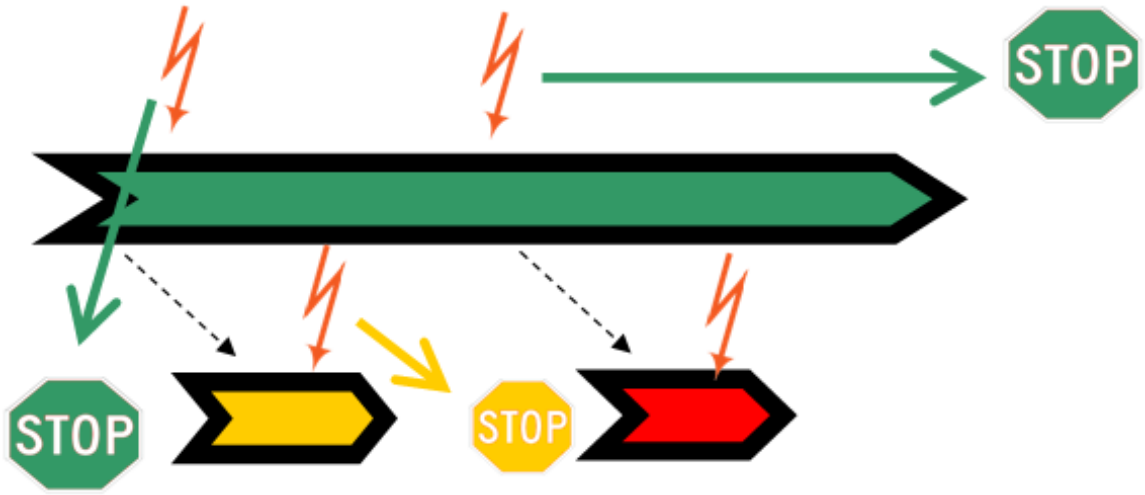
# User Manual: Test Automation Unit
# Version 1.15 for
# Testwell CTC++




| | |
|---|---|
| Version: | 0.9.3 |
| Date: | 2014-06-09 |
| Status: | In Progress / Adapted to Tool **/** Presented / **Reviewed** / Final |
| Author: | Oscar Slotosch, Robert Reitmeier, Milena Velichkova |
| File: | TAU_UserManual.docx |
| Size: | 43 Pages |

Verifysoft TECHNOLOGY

History:

| Version | Date | Status | Autor | Change |
|---|---|---|---|---|
| 0.1 | 2012-02-11 | In Progress | Slotosch | Template created |
| 0.2 | 2013-09-17 | In Progress | Slotosch | Adapted structure |
| 0.3 | 2013-09-18 | In Progress | Slotosch | Added requirements |
| 04. | 2013-09-20 | In Progress | Velichkova | Refined architecture |
| 0.5 | 2013-10-01 | In Progress | Reitmeier | Refined requirements and added tests |
| 0.6 | 2013-10-12 | Reviewed | Wildmoser | Reviewed |
| 0.7 | 2013-10-17 | Reviewed | Slotosch | Improved & finalized |
| 0.8 | 2014-02-03 | Presented | Slotosch | Adapted to Testwell CTC++ |
| 0.8.1 | 2014-04-17 | Presented | Slotosch | Added detectable errors and some test details |
| 0.9 | 2014-05-19 | Reviewed | Olavi Poutanen | Reviewed and extended Test/TAU specific descriptions |
| 0.9.1 | 2014-05-25 | Reviewed | Oscar Slotosch | Reviewed CTC specific changes and made small corrections |
| 0.9.2 | 2014-06-05 | Reviewed | Oscar Slotosch | Integrated V&V findings concerning validation, see #57 |
| 0.9.3 | 2014-06-09 | Reviewed | Slotosch | New TAU (1.15) and improved test TAU test descriptions, due to #70,#71,#72,#75 and TCA /TAU ticket #275 |
| 1.0 | <Date> | Final | | Finalized document |

Contents


**1 Scope of this Document ..... 6**

**2 Glossary ..... 7**

**3 Architecture of TAU ..... 8**

3.1 General Artifacts of TAU ..... 8
3.1.1 Test Plan TextExecution.txt ..... 9
3.1.2 Tool Configuration tool_config.py ..... 10
3.1.3 TestRun Directory ..... 10
3.1.4 Build-Folder ..... 10
3.1.5 Start Command: runTests.bat ..... 10
3.1.6 Start Command: runTests.py ..... 11
3.1.7 Log File: runTests.log ..... 11
3.1.8 Test Result Data <PATH>.Tst.xml ..... 11
3.1.9 Test Report junit-noframes.html ..... 11
3.2 Specific TAU Requirements for Testwell CTC++ ..... 12

3.3 Inter-TAU ..... 14
3.4 Intra-TAU for Testwell CTC++ ..... 14
3.5 Checkutils.py ..... 14

**4 Requirements ..... 14**

4.1 Requirements from ISO 26262 ..... 14
4.2 Requirements from IEC 61508 ..... 15
4.3 Requirements from EN 50128 ..... 15
4.4 Requirements from DO-330 / DO-178 C ..... 15
4.5 General TAU Requirements ..... 17
4.6 Verification: Satisfaction of Standard Requirements ..... 18
4.6.1 ISO 26262 ..... 18
4.6.2 IEC 61508 ..... 18
4.6.3 EN 50128 ..... 19
4.6.4 IEC DO-330 / DO 178C ..... 19
4.7 Verification: Satisfaction of General TAU Requirements ..... 19
4.8 Verification: Satisfaction of the specific TAU requirement for Testwell CTC++ ..... 20
4.9 Validation of TAU for the Qualification of Testwell CTC++ ..... 20

**5 Prerequisites of TAU ..... 21**

**6 Using TAU ..... 21**

6.1 Installation of TAU ..... 21
6.2 Configurating TAU ..... 21
6.3 Overall Arrangements in the Test Suite ..... 24
6.3.1 The Test Cases ..... 24
6.3.2 Two Types of Test Cases ..... 26
6.4 Before Starting to Run Tests with CTCQKit ..... 28
6.5 Adjusting the CTCQKit ..... 28
6.5.1 Some Assumptions and Clarifications ..... 28
6.5.2 Check/Adjust tracer.c ..... 29
6.5.3 Fine-tune the tool_config_XXX[_target].py Scripts for your Context ..... 29
6.5.4 Building and Running the Tests at Host ..... 31
6.5.5 Building and Running the Tests at Target ..... 31
6.6 Starting Tests ..... 32
6.7 Running Tests on Target ..... 32
6.8 Verifying Tests ..... 33
6.9 Analyzing Tests ..... 34
6.10 Documenting Test Results ..... 34

**7 Extending and Testing the TAU ..... 34**

7.1 Adding new Requirements for more Error Detections ..... 34
7.2 Test Suite for the General TAU Requirements ..... 34
7.2.1 TC_AutomaticTests ..... 35
7.2.1.1 T_FailingNegativeTest ..... 35
7.2.1.2 T_NegativeTest ..... 35
7.2.1.3 T_NoTstPy ..... 35

7.2.1.4 T_Skip ..... 35
7.2.1.5 T_SourceSyntaxError ..... 35
7.2.1.6 T_TestError ..... 35
7.2.1.7 T_TestFail ..... 35
7.2.1.8 T_TestSkip ..... 35
7.2.1.9 T_TestSuccess ..... 35
7.2.1.10 T_TimeoutDetected ..... 35
7.2.1.11 T_TimeOutNotDetected ..... 35
7.2.1.12 T_ToolCrash ..... 36
7.2.1.13 T_TstPyRuntimeError ..... 36
7.2.1.14 T_TstPySyntaxError ..... 36
7.2.2 TC_ManualTests ..... 36
7.2.2.1 T_DirectoryTAU ..... 36
7.2.2.2 T_ExecutableRelict ..... 36
7.2.2.3 T_NoTstPy ..... 36
7.2.2.4 T_RunTestsBat ..... 36
7.2.2.5 T_TestDataFolder ..... 36
7.2.2.6 T_DataRelict ..... 36
7.2.2.7 T_TestPlan ..... 37
7.2.2.8 T_TestReportFolder ..... 37
7.2.2.9 T_TestRunFolder ..... 37
7.2.2.10 T_ToolConfig ..... 37
7.2.3 Anomalous working conditions ..... 37
7.3 Test Suite for the TAU Requirements for Testwell CTC++ ..... 37
7.3.1 Test: C99Features ..... 37
7.3.2 Test: DecisionCoverage ..... 38
7.3.3 Test: General/Condition ..... 38
7.3.4 Test: General/Decision ..... 38
7.3.5 Test: General/Function ..... 39
7.3.6 Test: General/MCDC ..... 39
7.3.7 Test: General/Multicondition ..... 40
7.3.8 Test: StatementCoverage ..... 40
7.3.9 Test: MCDC/* ..... 40

**8 Licences ..... 41**

**9 References ..... 41**

**10 Appendix: Verification and Validation Report of TAU ..... 42**

10.1 Generic Requirements for the TAU ..... 42
10.2 Specific TAU requirements for Testwell CTC++ ..... 42
10.2.1 Test C99Features ..... 42
10.2.2 Test DecisionCoverage ..... 42
10.2.3 Test General/Condition ..... 42
10.2.4 Test General/Decision ..... 42
10.2.5 Test General/Function ..... 43
10.2.6 Test General/MCDC ..... 43
10.2.7 Test General/Multicondition ..... 43

10.2.8 Test MCDC/* ..... 43
10.2.9 Test StatementCoverage ..... 43
10.3 Validation of the TAU ..... 43

# 1 Scope of this Document

This document describes the Test Automation Unit (TAU) for the Testwell CTC++. The TAU can be used for tool qualification within many safety standards, since it satisfies their requirements for the validation of the tools.

This user manual describes how to use the TAU and shows the compliance to the standards. Furthermore it contains a description of the test cases that verify the TAU's implementation.

This TAU for the Testwell CTC++ has been derived from a generic TAU by adapting the tool specific parts to test the Testwell CTC++ and to detect the potential errors that have to be identified as critical during the analysis of potential errors of the Testwell CTC++. The verification has been successfully. This approach and the availability of the tests allows the user of the TAU to extend the TAU, ofr example to detect more potential errors, thus reducing the Testwell CTC++ specific safety guidelines during the application of the Testwell CTC++.

The document is structured as follows:

- Section 3 describes the generic architecture of the TAU that supported the adaptation to the Testwell CTC++.
- Section 4 lists the requirement from important safety standards and shows that the TAU conforms to them, provided that the test cases of the TAU in Section 7.2 have been passed successfully.
- Section 5 list the prerequisites for using the TAU
- Section 6 describes how to use the TAU to test the Testwell CTC++.
- Section 7 shows how the TAU can be extended and how it has to be verified after the extension.
- Section 10 shows the results from the verification and validation of the TAU as defined in Section 7.

# 2 Glossary

This section defines technical terms used within this document.

| **term** | **definition** |
|---|---|
| Build folder | Directory that contains all artifacts of the test |
| *Check* | possibility to detect an error |
| *Error* | in this document used as “potential error” |
| *Feature* (model) element | A function/feature of the tool that can be tested |
| *Restriction* | possibility to avoid an error |
| SUT | Subject under test, the tested tool in the specified version and configuration |
| TAU | Test Automation Unit, this tool |
| test | atomic test with result PASS/FAIL/SKIP/ABORT in a directory |
| test directory | A directory containing one or more tests (directories) |
| Test (model) element | Representation of a test directory in the model including a test description that specifies it |
| test suite | structured set of single tests |
| test plan | list of test (directories) to be executed in a file |
| Test result | The result/output of a test case. It can be as the reference value or deviate from it |
| Test verdict | Analysis of the test result provides the following verdicts:<br>• PASS: test result as expected<br>• FAIL: test result not as expected<br>• ERROR: no test result produced<br>• SKIP: test nit executable, due to configuration conflict<br>Note that in a negative test the test result might deviate from reference value but the verdict is PASS. |
| Testrun directory | the directory that contains all build-folders of the execution of all tests in a test plan |
| Validation | Ensures that the requirements are appropriate. The TAU is validated by running and testing it. |
| Verification | Ensures that the TAU satisfied all it’s requirements. The TAU is verified by review. |

# 3 Architecture of TAU

This section describes the architecture of the TAU by defining the components, and the artifacts (Section 3.1) of it. It mainly consists of a generic, tool independent part, the so called Inter-TAU (Section 3.2) and a tool specific part, the so called Intra-TAU (Section 3.4).

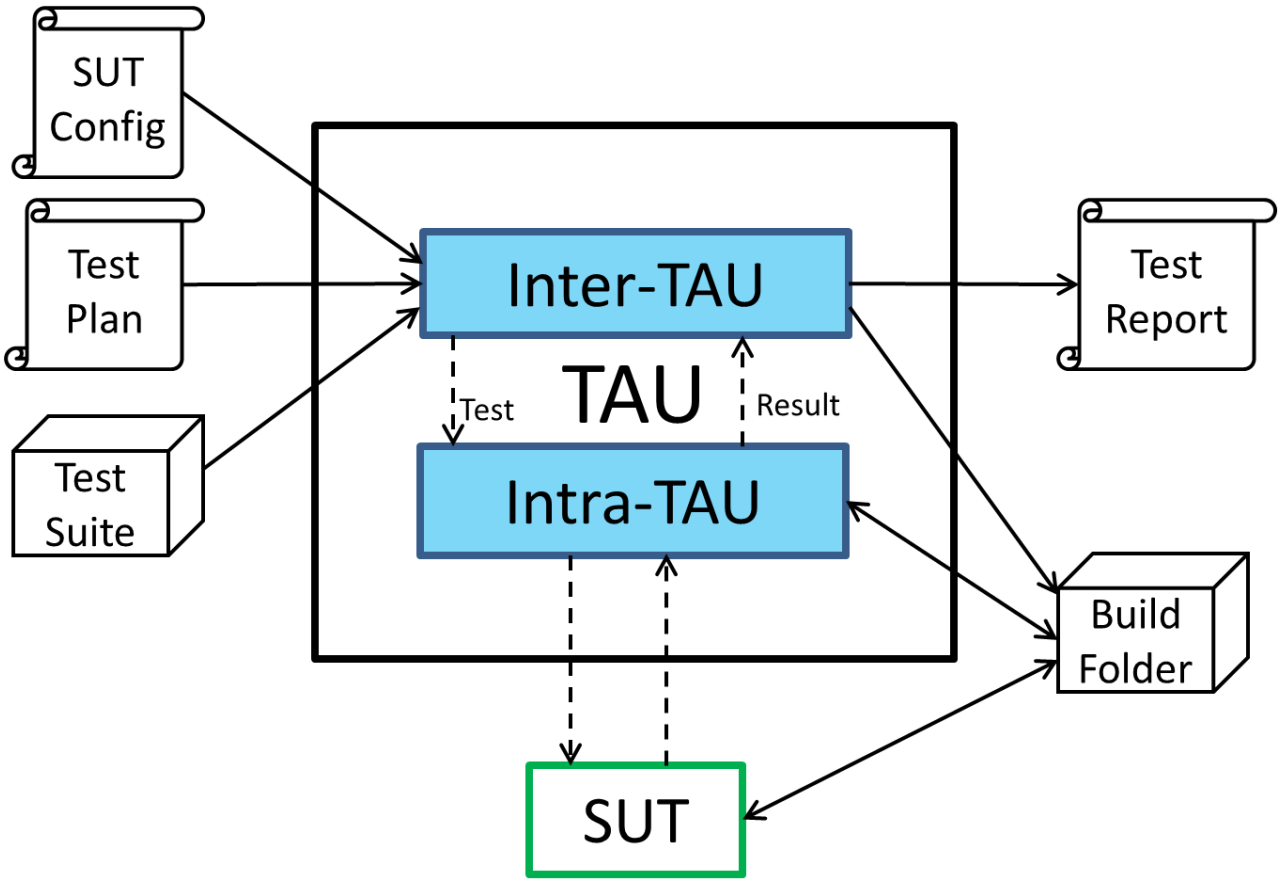



**Figure 1: Overview of TAU Architecture**

This architecture of the TAU is depicted in Figure 1. It shows the generic input and output artifacts and the TAU. The solid arrows represent the data flow between the components and artifacts, the dashed arrows the control flow. The main goal of the architecture is to encapsulate the tool specific parts into the Intra-TAU that executes a single test and to separate them from the generic test framework that is managing the tests and creating the overall report. The test plan is a list of directories in the test suite that are executed. The Inter-TAU creates a new directory, the so called "Build Folder" where the test is executed. Then it passes control to the Intra-TAU that runs the test by calling the SUT (Testwell CTC++). After the execution of the SUT the Intra-TAU computes the test result and returns it to the Inter-TAU. After the execution of the tests the Inter-TAU creates a test report.

## 3.1 General Artifacts of TAU

The general artifacts of the TAU are described here. The specific artifacts for tool Testwell CTC++ are described in Section 3.4.

### 3.1.1 Test Plan TextExecution.txt

The test plan is usually generated from the QST into the file <Qualification>/ Validation/ TestExecution.txt (where <Qualification> is the chosen qualification directory). Also other test plans can be used, for example to re-execute some tests that have been failing.
A test plan consists of a list of directories and comments (lines starting with a #).
The QST generates the test plan into the File TestExecution.txt

Furthermore the test plan contains the definition of the following variables:

- TS_ROOTDIRS: the root directory of the test suite.
- TESTRUN_ROOTDIR: the path to the TestRun directory (see Section 3.1.3).

After the keyword TESTS the test directories are listed (absolute or relative to the test suite root directory).

An example of a generated test plan is:

```
#
# This is a test plan generated from the Tool Chain Analyzer
#
# It covers 1 tool and 1 use case in 1 directories:
# - CTC
# - Target with Bitcov of CTC
#
TS_ROOTDIRS
<Testsuite>=C:\Programme\Qualification\CTCExample\QKit\Testsuite
#
TESTRUN_ROOTDIR
C:\Programme\Qualification\CTCExample\Validation\TestRun
#
TESTS
#
# DIRECTORY_NUMBER: 1
# DIRECTORY_PATH: <Testsuite>
#
<Testsuite>
#
# the directory contains the following TESTS:
# - 1. AllSize0
# - 2. AllSize1
# - 3. AllSize2
# - 4. C99Features
# - 5. Condition
# - 6. Decision
…
# - 46. XorSize3
#
# TEST_NUMBER: 1 / 1
# TEST_ID: CTC.Testsuite-MCDC-AllSize0
# TEST_NAME: AllSize0
# TESTED_FEATURE: ctc2html.bat
# TESTED_USECASE: Target with Bitcov
#
# TEST_DESCRIPTION
#   Tests in AllSize0
# END_TEST_DESCRIPTION
#
#
```

```
# TEST_NUMBER: 1 / 2
# TEST_ID: CTC.Testsuite-MCDC-AllSize1
# TEST_NAME: AllSize1
# TESTED_FEATURE: ctc2html.bat
# TESTED_USECASE: Target with Bitcov
…
```

**Figure 2: Generated Test Plan**

### 3.1.2 Tool Configuration tool_config.py

The tool configuration contains the configuration of the tool. The form of the concrete options, e.g. for compiling, linking and running depends on the Intra-TAU. InterTAU just takes the configuration as argument and passes it to IntraTAU in order to ensure that all test use the same configuration file.
The location of the tool configuration is <Qualification>/ Validation/ ToolConfig/tool_config.py (where <Qualification> is the chosen qualification directory). Also other test plans can be used.

### 3.1.3 TestRun Directory

Directory that contains all build folders and the other information of the test runs:

- Build Folders, see Section 3.1.4,
- runTests.bat, see Section 3.1.5,
- runTests.py, see Section 3.1.6,
- runTests.log, see Section 3.1.7 and
- TestData, see Section 3.1.8.

### 3.1.4 Build-Folder

The build folder contains all files required / generated from executing the tests (see Intra-TAU).

### 3.1.5 Start Command: runTests.bat

The main script to run the test is runTests.bat. It is a Windows script that contains the settings, creates and calls runTests.py.
It contains the following variable settings that need to be checked / adapted before the tests are started

- QUALIFICATION_DIR: The qualification directory
- JAVA_HOME: the path to the used Java jre
- PYTHONPATH: the path for python to search for modules
- TEST_PLAN: the test plan to execute
- CONFIG_FILE the configuration of the SUT / tool

Usually it suffices to configure JAVA_HOME, since the others are usually contained in the qualification directory and passed via arguments.
The tests should be started as follows:

```
./runTests.bat TestExecution.txt ToolConfig\tool_config.py > runTests.log 2>1 &
```

This creates the log file that documents the test run (see Section 3.1.7).

### 3.1.6 Start Command: runTests.py

It is generated and automatically started from the runTests.bat command, hence it should not be changed.

### 3.1.7 Log File: runTests.log

It is generated from running the tests, if they are started as described in Section 6.6.
The log file contains starting and ending time, the chosen configuration and the outputs from the test execution. For each test case the test log contains messages of the following form:

- ERROR
- FAILURE

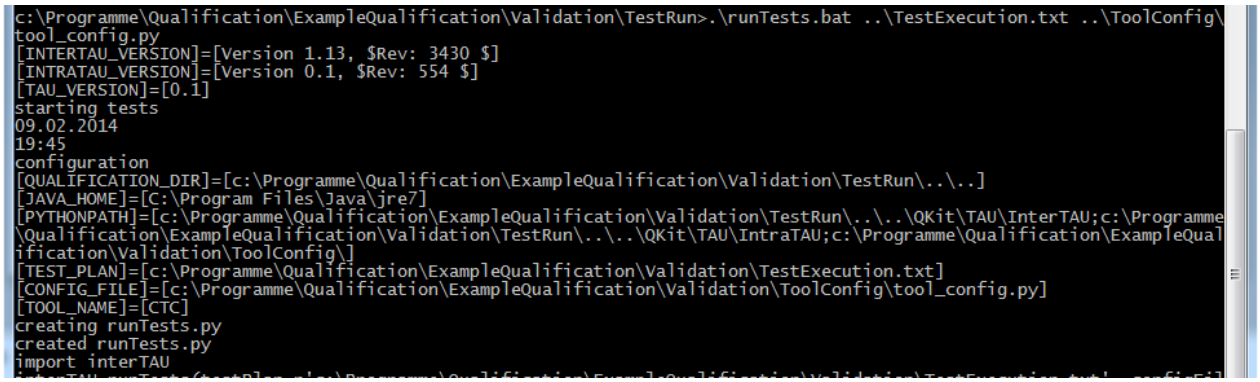



**Figure 3: Generated runTests.log**

### 3.1.8 Test Result Data <PATH>.Tst.xml

For every test there is a test result file of the form TestData/<PATH>.Tst.xml, where <PATH> is the path to the test and "/" are replaced by ".".

### 3.1.9 Test Report junit-noframes.html

The overall test report junit-noframes.html is contained in the directory <Qualification>/Validation/TestRun/TestReport.

**Summary**

| Tests | Failures | Errors | Success rate | Time |
|---|---|---|---|---|
| 110 | 30 | 0 | 72.73% | 439.916 |

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

**Package**

| Name | Tests | Errors | Failures | Skipped[0/1] | Time(s) |
|---|---|---|---|---|---|
| General | 3 | 0 | 1 | 0 | 1.482 |
| MCDC\AllSize0 | 4 | 0 | 0 | 0 | 1.326 |
| MCDC\AllSize1 | 4 | 0 | 2 | 0 | 1.872 |
| MCDC\AllSize2 | 4 | 0 | 2 | 0 | 11.451 |
| MCDC\EqSize0 | 4 | 0 | 0 | 0 | 1.482 |
| MCDC\EqSize1 | 4 | 0 | 2 | 0 | 1.654 |
| MCDC\EqSize2 | 4 | 0 | 2 | 0 | 4.307 |
| MCDC\EqSize3 | 4 | 0 | 2 | 0 | 33.951 |

**Figure 4: Generated Test Report**

The test report contains the test verdicts of the executed tests. The verdicts are interpreted as follows:

- PASS: The Test Report provides the message “Success” for the test case and the test log does not report any ERROR or FAILURE messages regarding this test case.
- FAIL: The Test Report provides the message “Failure” and the test log does not report any ERROR messages for the test case.
- ERROR: The Test Report provides the message “Error” for the test case, or the test log contains an Error message ERROR for the test case, or the test case is not contained in the Test Report at all.

SKIP: The Test Report marks the test case as skipped in the overview section.

## 3.2 Specific TAU Requirements for Testwell CTC++

In addition to the generic test requirements the TAU for Testwell CTC++ has to demonstrate the absence of the critical (unmitigated) potential errors.
They are listed in the Tool Qualification Plan in Section 6.7.1. To achieve a list of all errors that have to be detectable a configuration has been created with all testable features and without safety guidelines. The resulting list of critical errors is depicted in Figure 5.

### 6.7.1 Critical Errors in CTC

The following potential errors with TD 2 (medium mitigation probability) or TD 3 (low mitigation probability) have been identified and are listed in this section:

- C Code Coverage too Low, see Table 1
- C Code has Wrong Behaviour, see Table 2
- C Code MC/DC Coverage too High, see Table 3
- C Code Other Coverage too High, see Table 4
- Condition Coverage Too High, see Table 5
- Condition Coverage Too Low, see Table 6
- Coverage Data Corrupted, see Table 7
- Coverage Data Corrupted, see Table 8
- Coverage too Low, see Table 9
- Decision Coverage Too High, see Table 10
- Decision Coverage too High, see Table 11
- Decision Coverage Too Low, see Table 12
- Decision Instrumentation not Working, see Table 13
- Function Coverage Too High, see Table 14
- Function Coverage too High, see Table 15
- Function Coverage Too Low, see Table 16
- Function Instrumentation not Working, see Table 17
- Generating Execution Profile, see Table 18
- MC/DC Coverage Too High, see Table 19
- MC/DC Coverage Too High, see Table 20
- MC/DC Coverage too High, see Table 21
- MC/DC Coverage Too Low, see Table 22
- MC/DC Coverage Too Low, see Table 23
- MultiCondition Coverage Too High, see Table 24
- Multicondition Coverage too High, see Table 25
- MultiCondition Coverage Too Low, see Table 26
- Multicondition Instrumentation not Working, see Table 27
- Other Coverage too High, see Table 28
- Reducing to Condition Coverage, see Table 29
- Reducing to Decision Coverage, see Table 30
- Reducing to Function Coverage, see Table 31
- Reducing to MC/DC Coverage, see Table 32
- Statement Coverage Too High, see Table 33
- Statement Coverage Too Low, see Table 34
- Wrong Behaviour, see Table 35
- Wrong Behaviour, see Table 36
- Wrong Behaviour, see Table 37
- Wrong Behaviour, see Table 38
- Wrong Behaviour, see Table 39
- Wrong Behaviour, see Table 40
- Wrong Behaviour, see Table 41

**Figure 5: Potential Errors in testable features of Testwell CTC++**

Note that some of the mentioned errors can occur in different features of CTC (e.g. wrong behaviour).

### 3.3 Inter-TAU

The Inter-TAU processes the test plan, creates a build directory in the specified test run directory, runs the test execution and generates the test report. If a directory contains sub directories that contain nested test cases than those tests are also executed. Note that the Inter-TAU recognizes a test by the presence of a Tst.py file.

The Inter-TAU requires the following Software components:

- Python, since it is written in Python
- ANT and ANT Contrib: to generate the html report
- Java to run ant

ANT and ANT Contrib are distributed together with the TAU, while JRE and Python need to be installed from the user (see Section 6.1).

The InterTAU uses some help routines that are in the Checkutils.py module, see Section 3.5.

### 3.4 Intra-TAU for Testwell CTC++

The Intra-TAU contains methods for running compilation and simulation jobs in order to produce the outputs to be tested. These methods use the tool configuration file (see Section 3.1.2) to construct a command line call and then they run it. The methods are in turn called from a test setup method in each Tst.py. Note that the current build directory is being renewed (delete old and create new one) before each compilation job run.
The Intra-TAU executes every CTC test case. The all have a similar scheme, but may vary based on the configuration of the user.
The general steps are:

- Preparation: e.g. configuration of the test
- Instrumentation and compilation
- Execution
- Test Analysis: creation of test data and analysis
- Creation of Coverage Report

More details can be found in the File intraTAU.ph

### 3.5 Checkutils.py

The module checkutils.py inside the InterTAU directory provides some tool independent methods which are used in the test-methods in the Tst.py modules to compute a verdict, e.g. does a string contain a given pattern, has an output file been produced, etc.

# 4 Requirements

This section describes the requirements of the TAU, i.e. the goals that shall be achieved from the TAU.

### 4.1 Requirements from ISO 26262

The ISO 26262 [ISO26262] has the following requirements for Tool Qualification by Validation (Part 8, Section 11.4.9) Validation:

- The validation of the software tool shall meet the following criteria:

- ISO-8-11.4.9.a: the validation measures shall demonstrate that the software tool complies with its specified requirements
- ISO-8-11.4.9.b: the malfunctions and their corresponding erroneous outputs of the software tool occurring during validation shall be analysed together with information on their possible consequences and with measures to avoid or detect them, and
- ISO-8-11.4.9.c: the reaction of the software tool to anomalous operating conditions shall be examined

## 4.2 Requirements from IEC 61508

The IEC 61508 [IEC61508] has the following requirements on tool validation in Section 7.4.4.7: The results of tool validation shall be documented covering the following results:

- IEC-61508-7.4.4.7.a: a chronological record of the validation activities;
- IEC-61508-7.4.4.7.b: the version of the tool product manual being used;
- IEC-61508-7.4.4.7.c: the tool functions being validated;
- IEC-61508-7.4.4.7.d: tools and equipment used;
- IEC-61508-7.4.4.7.e: the results of the validation activity; the documented results of validation shall state either that the software has passed the validation or the reasons for its failure;
- IEC-61508-7.4.4.7.f: test cases and their results for subsequent analysis;
- IEC-61508-7.4.4.7.g: discrepancies between expected and actual results

## 4.3 Requirements from EN 50128

The EN 50128 [EN50128] has the following requirements on tool validation in Section 6.7.4.5: The results of tool validation shall be documented covering the following results:

- EN-6.7.4.5.a: a record of the validation activities;
- EN-6.7.4.5.b: the version of the tool manual being used;
- EN-6.7.4.5.c: the tool functions being validated;
- EN-6.7.4.5.d: tools and equipment used;
- EN-6.7.4.5.e: the results of the validation activity; the documented results of validation shall state either that the software has passed the validation or the reasons for its failure;
- EN-6.7.4.5.f: test cases and their results for subsequent analysis;
- EN-6.7.4.5.g:discrepancies between expected and actual results.

## 4.4 Requirements from DO-330 / DO-178 C

The DO-178 C [DO178C] determines the tool qualification level (TQL) depending on the determined tool criteria and the software level by using the following table:

| Software Level | Criteria | | |
|---|---|---|---|
| | 1 | 2 | 3 |
| A | TQL-1 | TQL-4 | TQL-5 |
| B | TQL-2 | TQL-4 | TQL-5 |
| C | TQL-3 | TQL-5 | TQL-5 |
| D | TQL-4 | TQL-5 | TQL-5 |

**Figure 6: TQL Determination according to DO-178 C**

The requirements on tool qualification (tool testing) are described in the DO-330 [DO330] in two parts: one for the verification tests (see Section 6.1.4 Tool Testing) from the tool provider against the tool requirements and one for the tests in the operational environment of the user (Section 6.2.2.c: Verification and validation in tool operational environment) against the tool operational requirements.

The TAU can be used for the development and the operational tests.
Therefore it has to comply with the following requirements from DO-330:

- DO-6.1.4.2.a: Test cases and procedures should be developed to demonstrate that the tool satisfies its requirements:
  - DO-6.1.4.2.a.1: Each test case is developed from the requirements (either Tool Requirements or low-level tool requirements) and identifies the set of inputs, the conditions, the expected results, and the pass/fail criteria.
  - DO-6.1.4.2.a.2: Test procedures are generated from the test cases
  - DO-6.1.4.2.a.3: Trace Data is generated between the test cases and the test procedures
- DO-6.1.4.2.b: Normal range tests should be performed, including the following:
  - DO-6.1.4.2.b.1: Real and integer input data should be exercised using valid equivalence classes and boundaries values.
  - DO-6.1.4.2.b.2: For state transitions, tests cases should be developed to exercise the transitions possible during normal operation
  - DO-6.1.4.2.b.3: For requirements expressed by logic equations or for requirements that include input logic combination, the normal test cases should verify the variable usage and the Boolean operators. In particular, for TQL-1, the tests cases should show that each requirement condition independently affects the required outcome by varying just that requirement condition while holding fixed all other possible requirement conditions.
- DO-6.1.4.2.c: Robustness tests should be performed to address all failure modes (for example, anomalous activation modes, inconsistency inputs, etc.) identified in Tool Requirements
- DO-6.1.4.2.d: If necessary, additional robustness tests should also be developed to complete the demonstration of the following:
  - DO-6.1.4.2.d.1: The ability of the tool to respond to anomalous inputs or conditions

    - DO-6.1.4.2.d.2: The detection of anomalous behavior
    - DO-6.1.4.2.d.3: The prevention of invalid output
- DO-6.1.4.2.e: Requirements-based test coverage analysis should be performed to demonstrate that all requirements (Tool Requirements and low-level tool requirements) have been tested. This analysis ensures that each requirement has at least one test case and that appropriate normal and robustness testing has been performed

- DO-6.6.6-c.1: Test cases and procedures should be developed to ensure that the tool satisfies the Tool Operational Requirements. These test cases and procedures should be requirements-based
- DO-6.6.6-c.2: Test procedures should be executed after the installation of the Tool Executable Object Code in the tool operational environment
- DO-6.6.6-c.3: Test results should be reviewed to ensure that test results are correct and that discrepancies between actual and expected results are explained
- DO-6.6.6-c.4: In order to validate the Tool Operational Requirements the tool should be exercised. The set of inputs selected should represent the actual tool use and its interfaces in the tool operational environment. This validation should confirm that the Tool Operational Requirements are correct and complete

## 4.5 General TAU Requirements

The TAU has the following general requirements directly derived to satisfy the standard requirements

- TAU-001: TAU shall be able to record the start and end time of the executed tests.
- TAU-002: The TAU shall generate a test report that allows to analyze the test results.
- TAU-003: The TAU shall support different versions and configurations of the validated tool. The tool configuration shall be specifiable in a global file.
- TAU-004: The TAU shall be able to detect and skip test cases that do not fit to the actual tool configuration.
- TAU-005: The TAU shall be able to document the test results including the used version of the TAU.
- TAU-006: The TAU shall create a directory, the so called “Build-Folder” for each executed test.
- TAU-007: The TAU shall support specification of a folder, the so called “TestRun-Folder” that contains all Build-Folders.
- TAU-008: The TAU shall compute the following test verdicts:
    - PASS: test result as expected
    - FAIL: test result not as expected
    - ERROR: no test result produced
    - SKIP: test not executable due to configuration conflict

  and provide a summary report.
- TAU-009: The TAU shall generate a log file that allows to reproduce the test results by documenting the used settings.
- TAU-010: The TAU shall produce outputs that allow to analyze and verify the produced test results.
- TAU-011: The TAU shall document the tool version and configuration for every test run.

- TAU-012: The TAU shall be able to execute tests under anomalous working conditions for itself and the tool.
- TAU-013: The TAU shall be able to deal robustly with the validated tool.
- TAU-014: The TAU shall be able to deal robustly with test implementations.
- TAU-015: The TAU shall be able to carry out negative tests.

To comply with the qualification model the TAU shall satisfy the following requirements:

- TAU-100: The TAU shall be able to execute test cases that are specified in a list of directories in a file, the so called test plan.
- TAU-101: The TAU shall ignore all lines starting with # in the test plan
- TAU-102: The TAU shall be able to execute directories that contain several tests in subdirectories.
- TAU-103: The TAU shall recognize tests by the existence of a Tst.py file in the directory or subdirectory.

Derived implementation requirements:

- TAU-201: The TAU shall be implemented in Python.
- TAU-202: The TAU shall have two parts: Inter/Intra.
- TAU-301: The Inter-TAU should contain all features that are independent from the tested tool.
- TAU-401 The Intra-TAU should contain the tool dependent features.
- TAU-402 The Intra-TAU shall be able to detect if the potential errors occur.

The TAU for Testwell CTC++ should be able to detect the following potential errors (described in the model of the tool for Testwell CTC++):

## 4.6 Verification: Satisfaction of Standard Requirements

Verification of the TAU requirements is accomplished by showing that the requirements of the TAU are sufficient for the standards. This is the case, if all requirements from the safety standards are covered by the TAU requirements. The TAU for Testwell CTC++ satisfies the requirements of the standards as follows:

### 4.6.1 ISO 26262

The requirements of [ISO26262] listed in Section 4.1 are satisfied by the TAU requirements as follows:

- ISO-8-11.4.9.a: The requirement of the TAU is to show the absence of the potential errors in the tool (“Tool complies with its specified requirements”). Those potential tool errors are listed in the previous section using the IDs Error-N, where N is a number starting with 001.
- ISO-8-11.4.9.b: TAU records test times (TAU-001), computes test results (TAU-008) and allows to analyze them (TAU-010 and TAU-011)
- ISO-8-11.4.9.c satisfied by TAU-012 and TAU-010.

### 4.6.2 IEC 61508

The requirements of [IEC61508] listed in Section 4.2 are satisfied from the as follows:

- IEC-61508-7.4.4.7.a satisfied by TAU-001
- IEC-61508-7.4.4.7.b is not applicable to the TAU, but to the Tool Identification in the Tool Qualification Plan. We hereby state that this requirement shall be fulfilled.

- IEC-61508-7.4.4.7.c satisfied by TAU-003
- IEC-61508-7.4.4.7.d is not applicable to the TAU, but to the Tool Qualification Report. We hereby state that this requirement shall be fulfilled.
- IEC-61508-7.4.4.7.e satisfied by TAU-002, TAU-005, TAU-008.
- IEC-61508-7.4.4.7.f satisfied by TAU-002, TAU-003
- IEC-61508-7.4.4.7.g satisfied by TAU-005, TAU-008

### 4.6.3 EN 50128

The requirements of [EN50128] listed in Section 4.3 are equivalent to the requirements of [IEC61508], hence they follow from the TAU requirements along the same lines (see 4.6.2).

### 4.6.4 IEC DO-330 / DO 178C

The requirements of [DO330] listed in Section 4.4 are requirements on the test suite (and its development) that is used for tool validation. The used test suite has to fulfill the listed requirements, and this has to be documented in the Test Qualification Plan.

## 4.7 Verification: Satisfaction of General TAU Requirements

All of the TAU requirements are satisfied by the TAU, which is shown in the following:

- TAU-001 (recording start and stop times) is satisfied because in the TAU log file the start and end times are recorded, see section 6.8 Verifying Tests and test case TC_TestSuccess.
- TAU-002 (test report analysis) is satisfied because the TAU does generate a test report.
- TAU-003 (different tool versions and configurations) is satisfied because one can specify a tool_config file and, inside it, the tool path, its command line arguments/options.
- TAU-004 (skip tests) is satisfied because in the Tst.py script one can encode conditions under which a test shall [not] be executed.
- TAU-005 (test results and TAU version) is satisfied because all test results and TAU's version are documented in the test report.
- TAU-006 (build folder) is satisfied because the TAU creates a such a folder for each test in the test run directory.
- TAU-007 (test run folder specification) is satisfied because in the test plan the TESTRUN_ROOTDIR can be specified.
- TAU-008 (test verdicts) is satisfied because for each test, the TAU computes a test verdict and a summary report in the test report.
- TAU-009 (log file) is satisfied because the TAU prints the used settings and one can have TAU create a log file by redirecting stdout and stderr into a log file. Moreover, the tool_config file is copied into the test run folder.
- TAU-010 (outputs fit for analysis and verification) is satisfied because the TAU produces a test report, a log file and tool output files.
- TAU-011 (tool version and configuration) this is done from the tau and tested in the test suite of the TAU.
- TAU-012 (anomalous conditions) is satisfied because of the successful test in the test suite.
- TAU-013 (robustness with the tool) is satisfied because there is a tool invocation time-out and a crash can be detected (indirectly).

- TAU-014 (robustness with test implementations) is satisfied because the TAU will detect errors in the test implementations (see tests TC_TstPyRunTimeError and T_TstPySyntaxError).
- TAU-015 (negative tests) is satisfied because the negative test logic can be implemented in a Tst.py script (together with TAU-013).
- TAU-100 (test plan) is satisfied because one has to specify the test plan as command line parameter to the TAU main script.
- TAU-101 (test plan comment lines) is satisfied because it is possible to have comment lines starting with a hash ('#') in the test plan.
- TAU-102 (tests in subdirectories) is satisfied because in the test plan one can specify subdirectories containing test directories.
- TAU-103 (test recognition) is satisfied literally.
- TAU-201 (TAU in Python) is satisfied because all TAU source files are Python scripts.
- TAU-202: Is satisfied by construction of TAU.
- TAU-301 (independent Inter-TAU) is satisfied directly by the manual test 'T_DirectoryTAU'.
- TAU-401 (tool dependent Intra-TAU) is also satisfied directly by the manual test 'T_DirectoryTAU'.

## 4.8 Verification: Satisfaction of the specific TAU requirement for Testwell CTC++

For the specific tool Testwell CTC++ the following requirements arise to show that the tool does not have the potential errors:

- TAU-402 (potential errors) the TAU, this is true, since the Intra-TAU for Testwell CTC++ detects all critical errors of Testwell CTC++, mentioned in Figure 5 on page 13. This is demonstrated by execution of the test cases described in Section 7.3
    - All Coverage too High / Low errors are tested, since the coverage is compared using identity comparison in all mentioned tests for
        - Condition coverage, see Section 7.3.3
        - Decision coverage, see Section 7.3.4
        - Function coverage, see Section 7.3.5
        - Multicondition coverage, see Section 7.3.7
        - Statement coverage, see Section 7.3.8
        - MC/DC coverage, see Section 7.3.9
    - All Wrong Behaviour errors are detected, since the tests are executed twice (with/without instrumentation) and the results are compared
    - Coverage reduction errors are detected, since the coverage comparison is based on the reduced reports.

## 4.9 Validation of TAU for the Qualification of Testwell CTC++

The validation of the TAU needs to show that the TAU is doing the right things. This means that the classification of the analysis potential errors is complete and that the tool specific potential errors listed in Section 3.2 are those identified in the classification report and listed in the tool qualification plan for the Testwell CTC++.

# 5 Prerequisites of TAU

The TAU (Test Automation Unit) has been tested under Windows 7 (Service Pack 1) but can be used in all environments under which the required components are available. The following components are required for the TAU

- Python – http://www.python.org/download/releases/3.2/
- JRE - java www.java.com/download

The qualification contains the ANT and ANT Contrib files in the directory <Qualification>\QKit\TAU\GlobalRessources.

# 6 Using TAU

The TAU is delivered within the qualification kit. It is located in the directory <Qualification>\QKit\TAU.

## 6.1 Installation of TAU

The TAU is written in Python, so it requires a python to be installed and configured such that the TAU routines can be found. This is done as follows:

1) Install Java (if not installed)
2) Install Python 3.2
    a. You will need Windows Admin Rights for the installation. A corresponding Win7 dialog will pop up during installation if your user account does not have admin rights.
    b. Download a Python installer version 3.2.3 or greater, e.g. python3.2.3.msi
    c. In Windows Explorer right click on the following file and select “Install”: python3.2.3.msi
    d. Install Python to C:\Python32 (default location)
    e. Extend the Windows PATH variable on your system by adding the path C:\Python32 to the right.
3) Verify Python: Open a MS-DOS CMD Shell and type “python --version”l letter V). The response should be “Python 3.2.3” printed to the CMD Shell
4) Verify TAU: start a CMD shell in the InterTAU directory and type “interTAU.py” to check if the TAU Version is 1.15

## 6.2 Configurating TAU

In the main start program runTests.bat (see Section 3.1.5) the following paths have to be adopted / checked:

- QUALIFICATION_DIR: The qualification directory
- JAVA_HOME: the path to the used Java jre
- PYTHONPATH: the path for python to search for modules
- TEST_PLAN: the test plan to execute
- CONFIG_FILE the configuration of the SUT / tool

Usually it suffices to configure JAVA_HOME, since the others are usually contained in the qualification directory and passed via argument.

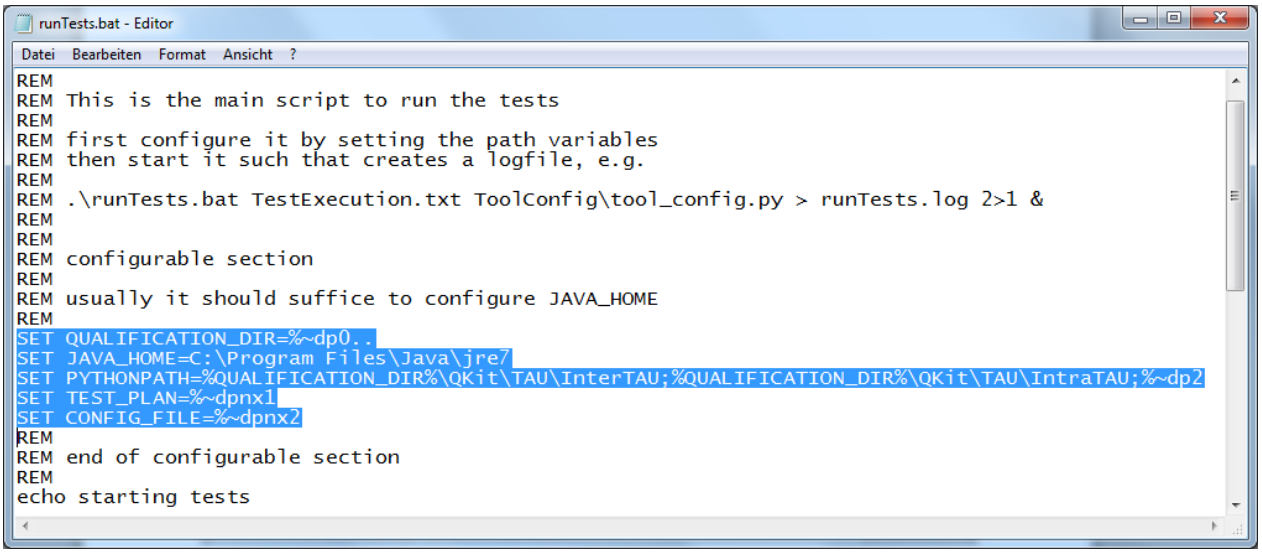



**Figure 7: Configuration of runTests.bat**

The main configuration of the tests is done in the file "tool_config.py". It contains the settings that can be changed, depending on the way CTC shall be qualified (see Figure 8).
There are several examples available in the ToolConfig directory that can be used. To use them just copy them e.g. by

*Copy tool_config_mcdc_target.py tool_config.py*

Note the default configuration provided is the decision coverage file, i.e. copy of tool_config_decision.py file.

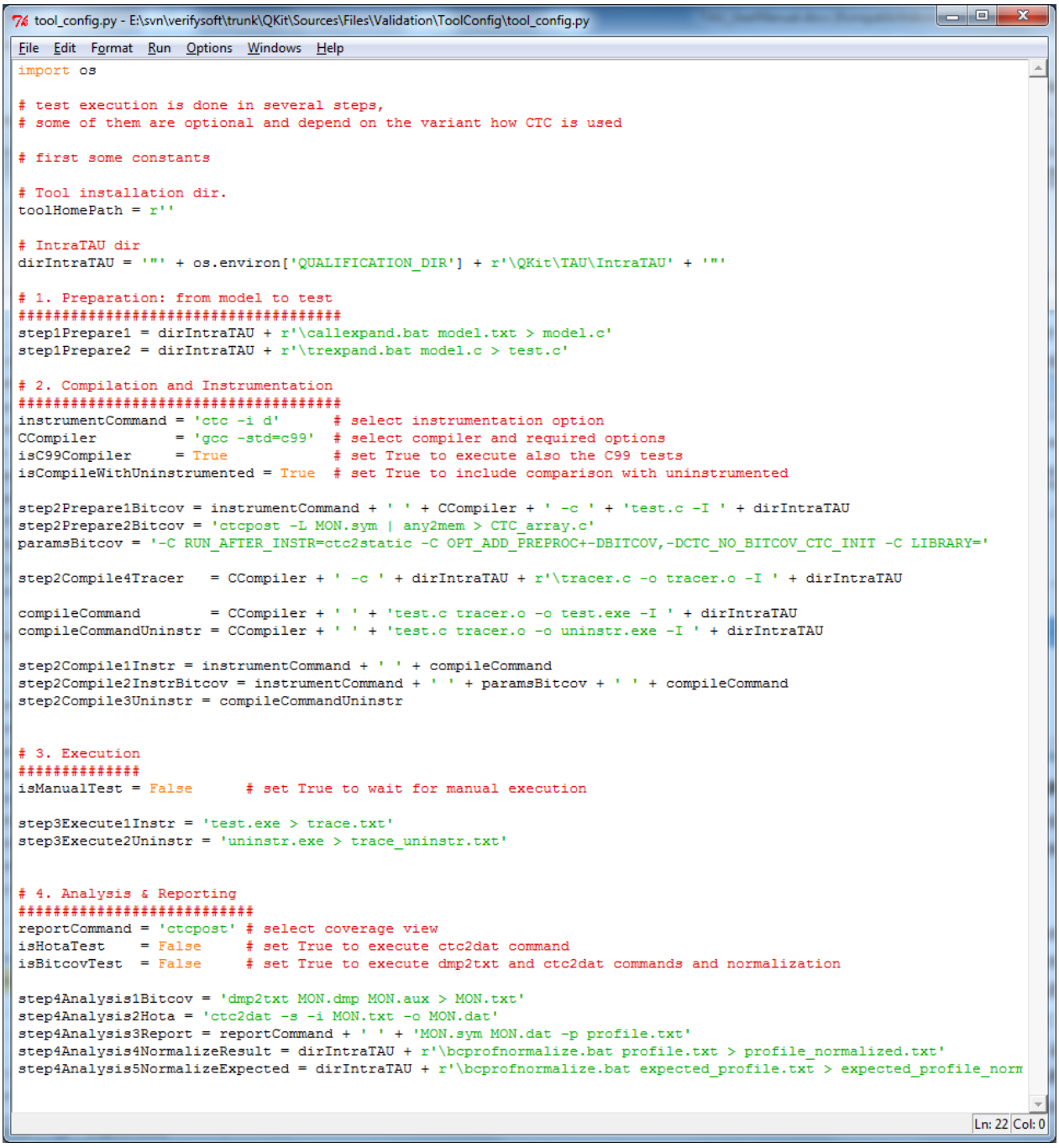

tool_config.py - E:\svn\verifysoft\trunk\QKit\Sources\Files\Validation\ToolConfig\tool_config.py
File Edit Format Run Options Windows Help
import os
# test execution is done in several steps,
# some of them are optional and depend on the variant how CTC is used
# first some constants
# Tool installation dir.
toolHomePath = r''
# IntraTAU dir
dirIntraTAU = '"' + os.environ['QUALIFICATION_DIR'] + r'\QKit\TAU\IntraTAU' + '"'
# 1. Preparation: from model to test
step1Prepare1 = dirIntraTAU + r'\callexpand.bat model.txt > model.c'
step1Prepare2 = dirIntraTAU + r'\trexpand.bat model.c > test.c'
# 2. Compilation and Instrumentation
instrumentCommand = 'ctc -i d'
CCompiler = 'gcc -std=c99'
isC99Compiler = True
isCompileWithUninstrumented = True
step2Prepare1Bitcov = instrumentCommand + ' ' + CCompiler + ' -c ' + 'test.c -I ' + dirIntraTAU
step2Prepare2Bitcov = 'ctcpost -L MON.sym | any2mem > CTC_array.c'
paramsBitcov = '-C RUN_AFTER_INSTR=ctc2static -C OPT_ADD_PREPROC+-DBITCOV,-DCTC_NO_BITCOV_CTC_INIT -C LIBRARY='
step2Compile4Tracer = CCompiler + ' -c ' + dirIntraTAU + r'\tracer.c -o tracer.o -I ' + dirIntraTAU
compileCommand = CCompiler + ' ' + 'test.c tracer.o -o test.exe -I ' + dirIntraTAU
compileCommandUninstr = CCompiler + ' ' + 'test.c tracer.o -o uninstr.exe -I ' + dirIntraTAU
step2Compile1Instr = instrumentCommand + ' ' + compileCommand
step2Compile2InstrBitcov = instrumentCommand + ' ' + paramsBitcov + ' ' + compileCommand
step2Compile3Uninstr = compileCommandUninstr
# 3. Execution
isManualTest = False
step3Execute1Instr = 'test.exe > trace.txt'
step3Execute2Uninstr = 'uninstr.exe > trace_uninstr.txt'
# 4. Analysis & Reporting
reportCommand = 'ctcpost' # select coverage view
isHotaTest = False
isBitcovTest = False
step4Analysis1Bitcov = 'dmp2txt MON.dmp MON.aux > MON.txt'
step4Analysis2Hota = 'ctc2dat -s -i MON.txt -o MON.dat'
step4Analysis3Report = reportCommand + ' ' + 'MON.sym MON.dat -p profile.txt'
step4Analysis4NormalizeResult = dirIntraTAU + r'\bcprofnormalize.bat profile.txt > profile_normalized.txt'
step4Analysis5NormalizeExpected = dirIntraTAU + r'\bcprofnormalize.bat expected_profile.txt > expected_profile_norm
Ln: 22 Col: 0

**Figure 8: Configuration Example**

In order to ease the configuration the qualification kit contains several example configurations (that all have been successfully tested during the release tests of the qualification kit):

- tool_config.py: the default example started from runTests.bat
- tool_config_decision[_target].py: a decision coverage example
- tool_config_condition[_target].py: a condition coverage example
- tool_config_function[_target].py: a function coverage example
- tool_config_mcdc[_target].py: a MC/DC coverage example
- tool_config_multicondition[_target].py: a multi-condition example

The tool_config_XXX.py variants are for host (assuming gcc compiler). The tool_config_XXX_target.py variants are for target (actually for target emulation at host using gcc compiler and gdb debugger, Bitcov instrumentation style is selected). In your real use case you need still to adjust these tool_config_XXX[_target].py a bit, e.g. what is your compiler and its options etc., and then copy and use the appropriate tool_config_XXX[_target] as tool_config.py).

## 6.3 Overall Arrangements in the Test Suite

When the QST tool is run a folder gets born, e.g. C:\Programme\Qualification-14\QKit\Testsuite. It contains some folders and subfolders. Each "leaf folder" is considered to be a test case in TAU sense. Each test case has a test program, which tests some property of CTC++. The TAU machinery builds the test program as uninstrumented and instrumented, runs them, compares the results and verifies the correctness of the behavior in various ways.

The test program is one file, test.c. It contains a number of test case functions with various C code usage situations, which we want to test, and main() function, which calls the test case functions.

The test program (test.c) can be instrumented for and run at the host or at the target machine. When "for the target machine", the instrumentation style can be "by Hota" or "by Bitcov". What instrumentation style is used and in what machine the tests are supposed to be run is determined by the settings there are in the tool_config.py file.

### 6.3.1 The Test Cases

There are the following test case directories with missions:

\StatementCoverage:
Tests which have been derived from C language syntax definition in systematic way. The purpose is to demonstrate that CTC++ handles properly the situations where in C a 'statement' can occur. Instrumentation is for decision coverage (-i d) and coverage report is "As instrumented". Active tool_config.py is one of tool_config_decision[_target].py.

\DecisionCoverage:
Tests which have been derived from C language syntax definition in systematic way. The purpose is to demonstrate that CTC++ handles properly the situations where in C a 'decision' can occur. Instrumentation is for decision coverage (-i d) and coverage report is "As instrumented". Active tool_config.py is one of tool_config_decision[_target].py.

\C99Features:
With C we mean the C99 level of language definition. Some C compilers, possibly also the one that you are using, do not implement all C99 level features. E.g. Microsoft Visual C++ (cl), when compiling C code, does not support the C99 features that are meant here. So that you anyway could run smoothly the StatementCoverage and DecisionCoverage test cases, the C99 level tests of them have been moved to this folder. Instrumentation is for decision coverage (-i d) and

coverage report is “As instrumented”. Active tool_config.py is one of tool_config_decision[_target].py. By default they are set to execute also the C99 tests. Specifically the C99 level things meant here are the following:

- variable declaration after statement, a’la … a = 5; int b; …
- variable declaration in for-loop, a’la for (int ii = 0; ii < 10; ii++) …

\General:
Actually this is split to 5 subfolders. In each of them there is same test program (test.c). In these tests the code is just instrumented differently by ctc.exe or the coverage report is taken with different coverage view by ctcpost.exe.

The test program is constructed by looking what the CTC++ User’s Guide says what C language constructs CTC++ claims to recognize and measure somehow. The purpose of these tests is to demonstrate that indeed CTC++ does what it promises when the central/normally used instrumentation options and coverage report taking options are used.

The subfolders are:

\General\Function:
The test program is instrumented for function coverage (-i f) and coverage report is “As instrumented”. Active tool_config.py is one of tool_config_function[_target].py.

\General\Decision:
The test program is instrumented for decision coverage (-i d) and coverage report is “As instrumented”. Active tool_config.py is one of tool_config_decision[_target].py.

\General\Multicondition:
The test program is instrumented for multicondition coverage (-i m) and coverage report is “As instrumented”. Active tool_config.py is one of tool_config_multicondition[_target].py.

\General\Condition:
The test program is instrumented for multicondition coverage (-i m) and coverage report is “Reduced to condition coverage” (-fc). Active tool_config.py is one of tool_config_condition[_target].py.

\General\MCDC:
The test program is instrumented for multicondition coverage (-i m) and coverage report is “Reduced to MC/DC coverage” (-fmcdc).Active tool_config.py is one of tool_config_mcdc[_target].py.

\MCDC:
The folder contains many (~25) subfolders, each testing some specific MC/DC coverage aspect. Instrumentation is for multicondition coverage (-i m) and coverage report is “Reduced to MC/DC coverage” (-fmcdc). Active tool_config.py is one of tool_config_mcdc[_target].py.

### 6.3.2 Two Types of Test Cases

Let's call these as a-type and b-type. All test cases are of a-type, except the \MCDC* test cases are of b-type. When the TAU machinery runs a test case of a-type or b-type the "testing game" goes a bit differently.

a-type test cases:

In an a-type test case folder there are initially the following files, prepared by the vendor:

- model.txt: is an easy to write, easy to read, high level description of what C constructs will be tested.
- expected_trace.txt: what trace the test program, either as when instrumented or as when uninstrumented, displays when run at host. At the target there should become similar trace, if only there were possible to do 'printf()' and capture that output to host file. Anyway, at the target test runs, the tracer module calculates CRC (32-bit word) of the trace calls (simpler to capture to host), and it anyway must be same as in the host test run.
- expected_profile.txt: the textual coverage report (when the selected instrumentation and reporting options of the test case have been used). At target tests the coverage report must become same (except run date etc. differences)
- gdbcmd.txt and gdbcmd_uninstr.txt: gdb debugger scripts that are used when running emulated target tests at host. When you go to your real who-knows-what target, these debugger scripts give you an idea what needs to be done when running the instrumented and uninstrumented programs at the target.
- Tst.py: the test case specific instructions to TAU machinery (in Python)

When the TAU machinery runs an a-type test case, the following happens:

- with callexpand.bat the model.txt file is converted to model.c file. The model.c is almost compilable C code, except it still has "S;" (~statement) and "Fi" (~function call returning value i) markings.
- with trexpand.bat the model.c is converted to compilable test.c C file. In this phase the "S;" and "Fi" markings are expanded to tracing calls with unique identification of the place of the call.
- C file tracer.c is compiled to tracer.o. tracer.c resides in …\QKit\TAU\intraTAU folder. You have adjusted its code so that it compiles in your context, and in minimum its CRC calculating code is preserved.
- an uninstrumented version of the test program is built comprising of the original test.c and of tracer.o. The resultant uninstrumented program is named to test.exe.
- test.c file is instrumented and an instrumented version of the test program is built comprising of the instrumented test.c and of tracer.o. The resultant instrumented program is named to itest.exe.
- the instrumented itest.exe is run. This run results file trace.txt and the collected coverage data in ctc-internal MON.dat form. If the test run is at the target, and depending if the instrumentation style is "Hota-way" or "Bitcov-way", there are some additional steps that needs to be done before MON.dat is obtained—discussed later…
- the uninstrumented test.exe is run. This run results file trace_uninstr.txt. If the test run is at the target, there are some additional steps that need to be done—discussed later…

- ctcpost is used to generate coverage report (comes under name profile.txt) from MON.sym (was born at instrumentation time) and from MON.dat (collected coverage data of the test run)
- files trace.txt (from instrumented itest.exe run) and expected_trace.txt (generated by the vendor) are compared for their sameness, if the test run at your site has been at host. If the test run at your site has been done at target, this check is not done. The reason is that at the target we cannot assume that the tracer.c module could have written such volumeous output of the program behavior (effectively some 'printf()' calls, which go to stdout, and output captured to host side to a text file). At target tests the trace.txt and trace_uninstr.txt are only one-line files (how to capture those files from the target is discussed later) and contain only the calculated CRC of the trace calls.
- files trace.txt and trace_uninstr.txt are compared for their sameness. If/when these files are same, it demonstrates that instrumentation of the code did not change the program behavior on what paths it executed. [The previous comparison of trace.txt and expected_trace.txt files is somewhat "nice to know", but, sure, those files should also be same]
- ctc2html is used to generate from profile.txt a HTML form coverage report
- a comparison is done (extracting some function and file level TER% lines from profile.txt and from its generated .html file) and thereby checked that HTML conversion of the coverage data did not come as biased.

Depending on how the TAU machinery has been configured (settings in active tool_config.py) some steps from the above scenario can be omitted. For example building/running of the uninstrumented program is not done (=> related checks are also not done). Or the HTML form report generation is not done. These possible variations are discussed later.

b-type test cases:

In a b-type test case folder there are initially the following files, prepared by the vendor:

- test.c: these files, in each MCDC* test case folder, are generated by a utility [not part of the QKit] and the program tests some MC/DC usage situations. Moreover, the generator utility has calculated and inserted to the file as C comment, e.g. /* ***TER 100 % ( 3/ 3) of FUNCTION T() */ a predicted value what the coverage TER% should be at each point of interest.
- expected_trace.txt: when the test program, either instrumented or uninstrumented, is run at host, it displays certain progress of its behavior and to the end how many of the called test functions passed and failed.
- gdbcmd.txt and gdbcmd_uninstr.txt: gdb debugger scripts that are used when running emulated target tests at host. When you go to your real who-knows-what target, these debugger scripts give you an idea what needs to be done when running the instrumented and uninstrumented program at the target. These debugger scripts are slightly different than in a-type tests (breakpoint is set differently and different variable is read out).
- Tst.py: the test case specific instructions to TAU machinery (in Python)

When the TAU machinery runs a b-type test case, basically the same happens as in a-type test cases, except the following differences:

- the starting point is straight away the test.c file. (That is, the model.txt --> model.c --> test.c steps are not done as in a-type test cases.)
- the test program (uninstrumented, instrumented) does not use the tracer.c module. [well, in each test case it is anyway compiled and linked to the test program, but because nobody calls it, it is actually in vain, but does not harm either…]
- if the test is run at host, the files expected_trace.txt (prepared by vendor) and trace.txt (written by the test program) are compared for their sameness.
- if the test program is run at target, the tracing information is reduced to one text line only (~to the bottom line of expected_trace.txt). That one-line information is captured from the target e.g. by debugger. When running instrumented program, the information comes to file trace.txt. When running uninstrumented program, the information comes to file trace_uninstr.txt. The TAU machinery compares these two files for their sameness.
- when the test has been run with instrumented program itest.exe, we eventually get profile.txt. From it (has the TER%s as CTC++ has calculated them) and from test.c (has the TER%s as the generator utility predicted them) the TER% lines are extracted and checked that CTC++ did report the coverage as predicted.

## 6.4 Before Starting to Run Tests with CTCQKit

The assumption is that in the beginning you have CTC++ fully functional at your context and you are familiar with CTC++ concepts and use. Your usage may be Host, Hota or Bitcov kind in the sense as discussed here before. For example,

- if your use case is for host (say, Visual C++ cl compiler, 64-bit code for Windows), you have the ctc.ini file fine-tuned appropriately, notably the CTC++ runtime library setting in place.
- if your use case is for target and Hota arrangement is used (say, some xcc cross compiler), you have adapted the Hota package targ*.c files to work in your target context and you have the ctc.ini file fine-tuned appropriately (the LIBRARY setting). And to you it is every-day practice how you get from the target context the collected encoded coverage data to host side to file MON.txt.
- if your use case is for target and Bitcov arrangement is used (say, some xcc cross compiler), it is every-day practice to you how you construct the CTC_array bit vector to the instrumented program and capture it to host side to MON.dmp file.

The mission is just to get CTC++ usage validated at your context using this CTCQKit.

## 6.5 Adjusting the CTCQKit

### 6.5.1 Some Assumptions and Clarifications

In this TAU machinery, in its various Python scripts how the CTC++ tests are done, it is assumed for example:

- The instrumentation is done like “ctc ctc-opts compiler compiler-opts-and-files”. The compiler may be invoked by path (some C:\PrFiles\XCC\bin\xcc.exe) or without (plain xcc or xcc.exe). In CTC++ there are also other ways to get ctc hooked itself to the compilation (some IDE integrations, “ctcwrap ctc-opts make …”, “ctcwrap –hard …”). The point is that they are not supposed to be used here, but exactly the above indicated way.

- The compiler, say “xcc”, both compiles and links. For example “xcc -o test.exe file.c tracer.o” kind of command is possible. If the compilation system that you use at your context requires separate command for compiling and for linking, you need to consult us how the Python scripts need to be modified (or perhaps you can do it by yourself, too)

### 6.5.2 Check/Adjust tracer.c

The a-type test cases use tracer.c. (It is also compiled and linked to b-type test cases, although in vain). This file is at …\QKit\TAU\intraTAU. Check this file that it works in your context.

Very likely you can use it out-of-the box. In such case, when it is compiled for host, it #includes <stdio.h> and makes some printf() calls to stdout. When the TAU machinery compiles it for target context, -DNO_FILE_IO flag is given, in which case the tracer reduces to “vanilla C” code, which only calculates the CRC of the trace calls (no ‘printf()’).

### 6.5.3 Fine-tune the tool_config_XXX[_target].py Scripts for your Context

These configuration scripts reside at …\Validation\ToolConfig. There are 10 configuration scripts available. They are a kind of models, which are already partially set up for their intended use with configuration attributes. One of them is the active one copied to tool_config.py at the folder.

The tool_config_XXX.py scripts are meant for host-based use. The XXX is one of function, decision, multicondition, condition and mcdc reflecting how the instrumentation is done and how the coverage report is taken.

Next it is assumed that you validate CTC++ in some target use case, and the cross compiler is ‘xcc’. So you need to look through all the 5 tool_config_XXX_target.py files, check and, if needed, to adjust settings:

Generated file extensions:
See the file extensions of tracer.o, test.exe, itest.exe. If your xcc compiler can work with these file extensions, fine. But if xcc would require, say tracer.obj instead of tracer.o, make the changes.

Compiler and its default options (CCompiler):
Initially there is ‘gcc -std=c99’. You need to change this to your compiler, some ‘xcc …’. Also give here the options that you use normally when compiling your production code, i.e. in what context you wish to validate CTC++. Such options could be your default optimization level, specification of target board architecture, etc.

Is xcc C99 compliant (isC99Compiler):
So, depending what xcc is, you select ‘True’ or ‘False’. In ‘False’ case the C99Features test case is not executed.

Is the uninstrumented program version built (isCompileWithUninstrumented):
If you select ‘True’, understand that both uninstrumented and instrumented program version is built and run. If the tests are run at the target, depending on your context, it may be work consuming to run a program (get image flashed, load to target, etc.).

If you select ‘False’, you lose the check that the program behaves similarly as uninstrumented and as instrumented. You anyway could put some weight on comparing the bottom lines of files expected_trace.txt (done by vendor and host context) and trace.txt (done by you at your target context)—e.g. in a-type tests the calculated CRC should be same.

Check that xcc compiler options are correct:
This means the tool_config_XXX[_target].py file settings step2Prepare1Bitcov, step2Compile4Tracer, compileCommand, compileCommandUninstr make sense to your xcc compiler and to your context where the test programs are compiled.

Are the test runs done “manually” or automatically by TAU machinery (isManualTest):

If you select ‘True’, the TAU machinery runs automatically the programs test.exe and itest.exe (at host or target as the case may be) and captures their trace files to trace_uninstr.txt and to trace.txt. If the instrumentation is for host, the itest.exe run also created the coverage data file MON.dat. The tool_config_XXX.py (for host) files have these settings already.

In the tool_config_XXX_target.py files there is default setting ‘True’ and the TAU machinery runs these two programs under gdb debugger control. That arrangement is used when running the target-built (Hota or Bitcov) programs at host as “host-emulated”.

Ok, but you have some who-knows-what target. If you can use similar debugger interface to run these programs at your target, you can keep this setting as ‘True’, but you need to adjust the debugger invocations step3Execute1Instr and step3Execute2Uninstr to your context. You also have to adjust the debugger command scripts gdbcmd.txt and gdbcmd_uninstr.txt accordingly to your context.

If you select ‘False’, the TAU machinery pauses for the time of running these two programs at the target. You have to run these programs “manually”, and capture same results to host side as the gdb debugger scripts would capture them. Once you have done that, you give to the TAU machinery permission to proceed (you hit ‘return’ key), and the TAU automation continues with test result checking.

Later there is discussion what the Hota-instrumented or Bitcov-instrumented target programs either itself need to produce to the host side, or what you need to capture of them somehow (e.g. by debugger) to the host side.

Settings isHotaTest and isBitcovtest:
Possible values are ‘True’ and ‘False’. Set these to reflect your way of instrumenting the target programs. Only the other of them can be ‘True’. If both are ‘False’, it means host-based instrumentation.

These settings have effect how the instrumentation is done. Specifically, if isBitcovTest==True, the TAU machinery first makes a “trial instrumentation” (see step2Prepare1Bitcov setting) to get MON.sym and then runs any2mem utility to get file CTC_array.c. Then the instrumentation is done again and into the test.c the just-generated file CTC_array.c is #included (contains a correct size CTC_array[] bit vector where to the “hit-bits” are recorded).

Setting isHTMLTest:
Default is 'True'. If you select 'False' the conversion from textual execution profile listing (profile.txt) to HTML representation is not done, and thereby the check that in HTML form there are same TER%s as in the profile.txt is not done.

### 6.5.4 Building and Running the Tests at Host

In this CTCQKit the test programs have been built for host with gcc compiler for Windows generating 32-bit code (gcc v4.8.1 from MinGW) using option '-std=c99'. You can repeat those tests, although they may not be very interesting, except for curiosity.

You however may have some other compiler, e.g. Microsoft Visual C++ (cl) compiler for Windows, with which you wish to qualify CTC++ use at Windows host. If so, you just fine-tune the tool_config_XXX.py scripts with compiler name and options etc., and the TAU machinery should run the tests out of the box using your compiler.

### 6.5.5 Building and Running the Tests at Target

The Hota and Bitcov style instrumented programs are meant here. The debugger scripts gdbcmd.txt and gdbcmd_uninstr.txt need to be adjusted for your debugger conventions (if you can run the target tests automatically from the host via debugger control). Or, if you have to run the test programs at the target manually, see below what you need to do.

If in the used tool_config_XXX.target.py script there is isCompileWithUninstrumented==False, test runs with uninstrumented program (test.exe) need not be done.

As mentioned earlier there are two types of tests, called as a-type and b-type. All test cases are of a-type, except the MCDC* test cases are of b-type. Another distinction is whether the instrumentation is using Hota or using Bitcov—in them the needed steps are slightly different.

When you look at the test.c program in all test cases (a or b type), there is code fragment:

```
#ifdef BITCOV
#include "CTC_array.c" /* contains generated, correct-size CTC_array[] */
#pragma CTC ENDSKIP
#else
   /* needed for technical reasons (esp. Hota), contents whatever... */
   unsigned char CTC_array[8] = {'n','o','t',' ','u','s','e','d'};
#endif
```

That is, also in Hota style instrumented programs there is CTC_array[] bit vector of some size. It is just a technical trick so that same gdbcmd.txt debugger script could be used both in Hota and in Bitcov target programs.

The test.exe and itest.exe programs are run at the target. When running these programs, you need to capture some information of the program behavior to the host side.

When running uninstrumented test program of an a-type test case:

A good place of the breakpoint is near the program end, at function display_crc() begin. At that point read unsigned int variable 'seed' and arrange its value to host to file trace_uninstr.txt in format "CRC=%u\n". And that's it.

When running uninstrumented test program of a b-type test case:

A good place of the breakpoint is near the program end, at function test_result() begin. At that point read int variable 'result' and arrange its value to host to file trace_uninstr.txt in format "Test result: All %d tests PASSED successfully.\n". And that's it.

When running instrumented test program, either a-type or b-type:

Do the same thing as when running the uninstrumented program, but at host side use file trace.txt to receive the 'seed' or 'result' variable.

Further, if the instrumentation is by Bitcov, at the same breakpoint capture the CTC_array[] bit vector (in C sense it is an unsigned char [] vector) to host side to file MON.dmp. When these are done, that's it.

When you look at the gdbcmd.txt debug scripts (which automate the above in some contexts), you see that the MON.dmp is captured also when the instrumentation is by Hota. It is actually in vain. The TAU machinery does not use it in Hota case. It is just captured because we can manage with same debug script both in Hota and Bitcov case.

Also understand that in Hota instrumented target programs, it is assumed that you have linked the Hota's targ*.c files to the executable, which writes the MON.txt file to the host side (or you get it to the host side in the same way as you do in your everyday practice to use Hota at your context). Further, if you look at the test.c programs, near their end you see line '#pragma CTC APPEND'. We have added it there just to help the arrangement, to ensure the triggering of the coverage data writing out at the program end.

## 6.6 Starting Tests

The tests can be started from a CMD-Shell in the directory: <Qualification>/Validation/TestRun (Where <Qualification> is the chosen qualification directory).

*.\runTests.bat ..\TestExecution.txt ..\ToolConfig\tool_config.py*

## 6.7 Running Tests on Target

If the tests are executed on the target there are several degrees of automation depending on the available test environment and their features.

The first and most important configuration setting is in the tool_config.py file (Section 4 Analysis):

*isBitcovTest = False*

If this is set to true the IntraTAU will use bitcov and the additional commands to convert the results.

Furthermore there are two commands in the tool_config.py that should be adopted to the target:

*step3Execute1Instr = 'itest.exe > trace.txt'*
*step3Execute2Uninstr = 'test.exe > trace_uninstr.txt'*

Furthermore there is a configuration option in the tool_config file:

isManualTest = False

if this is set to true the TAU will wait after every test until the user is finished and presses a key.

Some target support the possibility to use “printf” via some debugging tools and other’s don’t. In the case were no printf is supported the correct test execution cannot be compared using the printed values of the important variables, e.g. the “result” variable. In this case the tester has to verify the values of the variables using the debugger and compare them manually with the values in the reference files (which will lead to a failing test case that has to be analyzed in the qualification report) or he has to create the corresponding output files manually such that the evaluation runs successfully.

## 6.8 Verifying Tests

The successful execution of the tests can be verified by inspecting the log messages in the log file

- After starting: “STARTED TEST RUN”
- For each test in the test plan (or it’s contained tests): “Loaded <n> tests from <path>”, where <n> is the number of sub-tests and <path> the path to the executed test.
- At the end: “FINISHED TEST RUN” together with a duration that can be used to detect if the tests shave been running as expected.

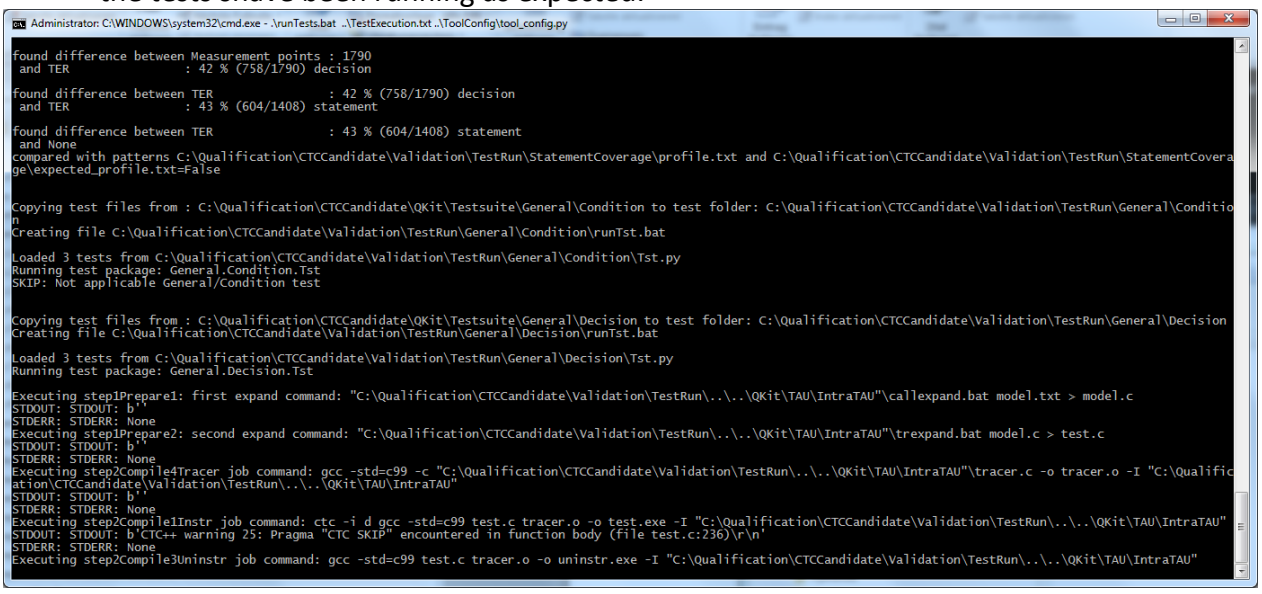

Administrator: C:\WINDOWS\system32\cmd.exe - .\runTests.bat ..\TestExecution.txt ..\ToolConfig\tool_config.py
found difference between Measurement points : 1790
and TER : 42 % (758/1790) decision
found difference between TER : 42 % (758/1790) decision
and TER : 43 % (604/1408) statement
found difference between TER : 43 % (604/1408) statement
and None
compared with patterns C:\Qualification\CTCCandidate\Validation\TestRun\StatementCoverage\profile.txt and C:\Qualification\CTCCandidate\Validation\TestRun\StatementCoverage\expected_profile.txt=False
Copying test files from : C:\Qualification\CTCCandidate\QKit\Testsuite\General\Condition to test folder: C:\Qualification\CTCCandidate\Validation\TestRun\General\Condition
Creating file C:\Qualification\CTCCandidate\Validation\TestRun\General\Condition\runTst.bat
Loaded 3 tests from C:\Qualification\CTCCandidate\Validation\TestRun\General\Condition\Tst.py
Running test package: General.Condition.Tst
SKIP: Not applicable General/Condition test
Copying test files from : C:\Qualification\CTCCandidate\QKit\Testsuite\General\Decision to test folder: C:\Qualification\CTCCandidate\Validation\TestRun\General\Decision
Creating file C:\Qualification\CTCCandidate\Validation\TestRun\General\Decision\runTst.bat
Loaded 3 tests from C:\Qualification\CTCCandidate\Validation\TestRun\General\Decision\Tst.py
Running test package: General.Decision.Tst
Executing step1Prepare1: first expand command: "C:\Qualification\CTCCandidate\Validation\TestRun\..\..\QKit\TAU\IntraTAU"\callexpand.bat model.txt > model.c
STDOUT: STDOUT: b''
STDERR: STDERR: None
Executing step1Prepare2: second expand command: "C:\Qualification\CTCCandidate\Validation\TestRun\..\..\QKit\TAU\IntraTAU"\trexpand.bat model.c > test.c
STDOUT: STDOUT: b''
STDERR: STDERR: None
Executing step2Compile4Tracer job command: gcc -std=c99 -c "C:\Qualification\CTCCandidate\Validation\TestRun\..\..\QKit\TAU\IntraTAU"\tracer.c -o tracer.o -I "C:\Qualification\CTCCandidate\Validation\TestRun\..\..\QKit\TAU\IntraTAU"
STDOUT: STDOUT: b''
STDERR: STDERR: None
Executing step2Compile1Instr job command: ctc -i d gcc -std=c99 test.c tracer.o -o test.exe -I "C:\Qualification\CTCCandidate\Validation\TestRun\..\..\QKit\TAU\IntraTAU"
STDOUT: STDOUT: b'CTC++ warning 25: Pragma "CTC SKIP" encountered in function body (file test.c:236)\r\n'
STDERR: STDERR: None
Executing step2Compile3Uninstr job command: gcc -std=c99 test.c tracer.o -o uninstr.exe -I "C:\Qualification\CTCCandidate\Validation\TestRun\..\..\QKit\TAU\IntraTAU"

**Figure 9: Example test output messages**

### 6.9 Analyzing Tests

For the analysis of the tests the test report and the build folders shall be used. For every found deviation in the Testwell CTC++ a mitigation (check or restriction) has to be described. Those have to be added to the tool qualification report and the tool safety manual of the Testwell CTC++.

### 6.10 Documenting Test Results

The test results (folder TestRun) should be documented, together with the used test suite.

# 7 Extending and Testing the TAU

This section describes how to maintain or extend the TAU. Whenever the TAU or its requirements are modified the following verification and validation steps need to be repeated in order to show that the TAU satisfies its requirements (requirements verification & test) and can be effectively used for its desired purpose of testing the absence of the right errors Testwell CTC++ (validation). An extension might be necessary if the validation of the TAU fails, e.g. if more potential errors should be detected that are currently supported from the TAU.

### 7.1 Adding new Requirements for more Error Detections

If the TAU for the Testwell CTC++ should be extended to detect more potential errors, this document need the following extensions:

- Extend the requirements list of the potential errors that TAU for Testwell CTC++ should detect in Section 3.2 as additional TAU-Requirements
- Validate that they comply with the requirements and document this in Section 10.3
- Extend the list of test cases that can detect the potential errors in Section 7.3
- Document the Verification of the additional TAU requirements by the test specification in Section 4.8.
- Extend the TAU for Testwell CTC++
- Verify that the TAU for Testwell CTC++ satisfies the requirements by running the tests in Section 7.2 and 7.3. Note that if only the Intra-Tau changes and the interfaces to the Inter-TAU remain unchanged it suffices to re-run the tests in Section 7.3.
- Document the verification tests results in Section 10.1 if applicable and in Section 10.2

### 7.2 Test Suite for the General TAU Requirements

The TAU is verified by checking the correct derivation of the TAU requirements (see Section 4.5) from the standards (see Section 4.6), by providing an argument for the satisfaction of each TAU requirement (see Section 4.7), by providing evidence for these arguments in form of tests by applying these tests to the TAU.
This section contains the test cases that demonstrate that the general requirements are satisfied. The test cases are grouped together in a test folder TF_InterTAU that is located in the development folder of the TAU. There are three test categories (sub-folders) contained that contain the single test cases:

- TC_AutomaticTests, see Section 7.2.1
- TC_ManualTests, see Section 7.2.2**Fehler! Verweisquelle konnte nicht gefunden werden.**
- TC_ManulaTest, see Section 7.2.3

The tests are described in the following subsections. More details can be found in the test implementations.

### 7.2.1 TC_AutomaticTests

This test categories contains automatic test cases, they can be executed automatically, just by adding the path to TC_AutomaticTests to the test execution list and running the TAU. It contains the following test cases:

#### 7.2.1.1 T_FailingNegativeTest

This test produces a negative test result and fails. The test verdict shall be “FAIL”.

#### 7.2.1.2 T_NegativeTest

This test produces a negative test result and checks for it. The test verdict shall be “PASS”.

#### 7.2.1.3 T_NoTstPy

This test produces no test result since the existence of Tst.py defines a test. The test verdict PASS has to be set manually, see Section 7.2.2.3.

#### 7.2.1.4 T_Skip

This test checks the skip functionality of the TAU, this test shall be skipped (as required by TAU-004). Test result shall be SKIP.

#### 7.2.1.5 T_SourceSyntaxError

This test checks the reaction of the TAU in the presence of an error during test execution, e.g. a syntax error during compilation, as required by TAU-013. Test result should be ERROR.

#### 7.2.1.6 T_TestError

This test checks the test verdict computation of the TAU in an error case, as required by TAU-008. The result of this test shall be ERROR.

#### 7.2.1.7 T_TestFail

This test checks the test verdict computation of the TAU in a failing case, as required by TAU-008. The result of this test shall be FAIL.

#### 7.2.1.8 T_TestSkip

This test checks the test verdict computation of the TAU in a skipped case, as required by TAU-008. The result of this test shall be SKIP.

#### 7.2.1.9 T_TestSuccess

This test checks the test verdict computation of the TAU in a success case, as required by TAU-008. The result of this test shall be PASS.

#### 7.2.1.10 T_TimeoutDetected

This test checks the TAU’s tool time-out feature, which is set to 5s. The tool would wait for 10s, but times out. The test is successful if the time-out occurs which can be verified by searching the standard output for the string “TIMEOUT”. In the case the timeout is detected this test should produce the result FAIL.

#### 7.2.1.11 T_TimeOutNotDetected

This contains a timeout that is not detected since the standard output is not searched for the string “TIMEOUT”. Therefore this test should produce the result PASS.

#### 7.2.1.12 T_ToolCrash

This test checks the reaction of the TAU when the tool crashes during test execution (reporting an error), as required by TAU-008 and TAU-013. Other crashes should be detected by missing/incomplete outputs. The test should be FAIL, since it checks for errors in stderr and then fails.

#### 7.2.1.13 T_TstPyRuntimeError

This test checks the reaction of the TAU in the presence of a runtime error in the python test implementation code (Tst.py), as required by TAU-014. The test verdict shall be “ERROR”.

#### 7.2.1.14 T_TstPySyntaxError

This test checks the reaction of the TAU in the presence of a syntax error in the python test implementation code (Tst.py), as required by TAU-014. This test is successful, if an error is mentioned for this test case in either in the test result file and/or in the log file. The test verdict shall be “ERROR”.

### 7.2.2 TC_ManualTests

This test categories contains manual test cases that have to be executed manually, e.g. by reviewing the TAU. It contains the following test cases:

#### 7.2.2.1 T_DirectoryTAU

This directory contains the implementation of the TAU. It is separated into an InterTAU and IntraTAU (as required by TAU-202). All TAU source files are Python source files (as required by TAU-201). None of the source files in InterTAU reference or import the tool_config.py module (as required by TAU-301), hence all tool dependencies are contained in IntraTAU (GlobalRessources does not contain TAU Python source code) (satisfying TAU-401).

#### 7.2.2.2 T_ExecutableRelict

This test checks if the TAU re-executes the test. This test has to executed twice:
This test satisfies TAU-013.

#### 7.2.2.3 T_NoTstPy

This test is executed automatically (see Section 7.2.1.3) but should not produce a test result since the existence of Tst.py defines a test. The test verdict PASS has to be set manually.

#### 7.2.2.4 T_RunTestsBat

The main test suite run script is runTests.bat, taking the test plan and the tool configuration file (satisfying TAU-003) as command line inputs. TAU creates a log file (thereby satisfying TAU-009 and TAU-011). This test should verify all these features. Also, this test verifies if the start and end time of the test suite are recorded in the log (TAU-001) and if it generates a report that contains all details of the test execution necessary to analyze the test result (TAU-002).

#### 7.2.2.5 T_TestDataFolder

In the TestData folder the TAU places documentation of each test in XML format. It contains the tool call command line and the tool’s standard and error output (as required by TAU-010).

#### 7.2.2.6 T_DataRelict

This test checks if the TAU re-executes the test and reused the data from the test suite. This test has to executed twice. This test satisfies TAU-013.

#### 7.2.2.7 T_TestPlan

In the test plan of the test suite we are testing the following features:

- For all specified test directories with a Tst.py file in it, TAU tries to execute the test (TAU-100),
- all lines starting with a hash ("#") are ignored (TAU-101), and

the test plan contains tests in subdirectories (TAU-102).

#### 7.2.2.8 T_TestReportFolder

The TAU will create a TestReport folder containing HTML documentation about the test suite run (as required by TAU-005 and TAU-008).

#### 7.2.2.9 T_TestRunFolder

In the specified TestRun folder (required by TAU-007), for each executed test a build folder is created (required by TAU-006) containing the tool output, the output artifact, and the simulation output (all required by TAU-010).

#### 7.2.2.10 T_ToolConfig

Checks that the file tool_config.py contains the appropriate configuration of the tool.

### 7.2.3 Anomalous working conditions

The test suite also has to be executed under anomalous working conditions as required by TAU-012. Among these are

- Heavy CPU load by starting other CPU intensive tasks (preferably other test suite runs)
- Very low primary and secondary memory by starting other tasks consuming all RAM and minimizing hard-disk space.

The test verdicts under each condition shall be equivalent to the verdicts of a successful test suite run under normal working conditions.

## 7.3 Test Suite for the TAU Requirements for Testwell CTC++

This section describes the tool specific test cases for Testwell CTC++.
The tool specific test cases can be taken from the Testsuite folder of the QKit. The TAU specific test cases are executed by starting the qualification support tool with a default configuration and automatically generating a qualification directory that contains the TAU and the tests.
The tests for Testwell CTC contain test that pass successfully and those that fail, e.g. due to too big expressions. Furthermore the tests compare the result of the execution with expected results and the un-instrumented code such that wrong behavior would be detected as well as coverage deviations that subsume all potential errors of CTC, see Section 4.8.

### 7.3.1 Test: C99Features

This test ensures that C99 features of StatementCoverage and DecisionCoverage tests work correctly. It also checks for the correct behavior. It can be started with the configuration file tool_config_decision.py, e.g. by:

```
copy /Y ..\ToolConfig\tool_config_decision.py ..\ToolConfig\tool_config.py
.\runTests.bat ..\TestExecution.txt ..\ToolConfig\tool_config.py
```

Of course the test has to be in the list of executed tests (TestExecution.txt).

It consists of the following sub-tests that are reported:

- testDiffOutFiles: checks if the execution is as expected and would detect wrong instrumentation
- testDiffUninstrumentedTrace: checks if the instrumented and uninstrumented code are identical, i.e. would detect “Wrong Behaviour”
- testDiffReportFiles: compare the coverage report with the expected coverage report and would detect errors like “Coverage too high” and “Coverage too low”
- testDiffTERinHTML

### 7.3.2 Test: DecisionCoverage

This test ensures that decision coverage is computed correctly. It also checks for the correct behavior. It can be started with the configuration file tool_config_decision.py, e.g. by:

*copy /Y ..\ToolConfig\tool_config_decision.py ..\ToolConfig\tool_config.py*
*.\runTests.bat ..\TestExecution.txt ..\ToolConfig\tool_config.py*

Of course the test has to be in the list of executed tests (TestExecution.txt).
It consists of the following sub-tests that are reported:

- testDiffOutFiles: checks if the execution is as expected and would detect wrong instrumentation
- testDiffUninstrumentedTrace: checks if the instrumented and uninstrumented code are identical, i.e. would detect “Wrong Behaviour”
- testDiffReportFiles: compare the coverage report with the expected coverage report and would detect errors like “Coverage too high” and “Coverage too low”
- testDiffTERinHTML

### 7.3.3 Test: General/Condition

This test ensures that condition coverage is computed correctly. It also checks for the correct behavior. It can be started with the configuration file tool_config_condition.py, e.g. by:

*copy /Y ..\ToolConfig\tool_config_condition.py ..\ToolConfig\tool_config.py*
*.\runTests.bat ..\TestExecution.txt ..\ToolConfig\tool_config.py*

Of course the test has to be in the list of executed tests (TestExecution.txt).
It consists of the following sub-tests that are reported:

- testDiffOutFiles: checks if the execution is as expected and would detect wrong instrumentation
- testDiffUninstrumentedTrace: checks if the instrumented and uninstrumented code are identical, i.e. would detect “Wrong Behaviour”
- testDiffReportFiles: compare the coverage report with the expected coverage report and would detect errors like “Coverage too high” and “Coverage too low”
- testDiffTERinHTML

### 7.3.4 Test: General/Decision

This test ensures that decision coverage is computed correctly. It also checks for the correct behavior. It can be started with the configuration file tool_config_decision.py, e.g. by:

*copy /Y ..\ToolConfig\tool_config_decision.py ..\ToolConfig\tool_config.py*
*.\runTests.bat ..\TestExecution.txt ..\ToolConfig\tool_config.py*

Of course the test has to be in the list of executed tests (TestExecution.txt).
It consists of the following sub-tests that are reported:

- testDiffOutFiles: checks if the execution is as expected and would detect wrong instrumentation
- testDiffUninstrumentedTrace: checks if the instrumented and uninstrumented code are identical, i.e. would detect "Wrong Behaviour"
- testDiffReportFiles: compare the coverage report with the expected coverage report and would detect errors like "Coverage too high" and "Coverage too low"
- testDiffTERinHTML

### 7.3.5 Test: General/Function

This test ensures that function coverage is computed correctly. It also checks for the correct behavior. It can be started with the configuration file tool_config_function.py, e.g. by:

*copy /Y ..\ToolConfig\tool_config_function.py ..\ToolConfig\tool_config.py*
*.\runTests.bat ..\TestExecution.txt ..\ToolConfig\tool_config.py*

Of course the test has to be in the list of executed tests (TestExecution.txt).
It consists of the following sub-tests that are reported:

- testDiffOutFiles: checks if the execution is as expected and would detect wrong instrumentation
- testDiffUninstrumentedTrace: checks if the instrumented and uninstrumented code are identical, i.e. would detect "Wrong Behaviour"
- testDiffReportFiles: compare the coverage report with the expected coverage report and would detect errors like "Coverage too high" and "Coverage too low"
- testDiffTERinHTML

### 7.3.6 Test: General/MCDC

This test ensures that MC/DC coverage is computed correctly. It also checks for the correct behavior. It can be started with the configuration file tool_config_mcdc.py, e.g. by:

*copy /Y ..\ToolConfig\tool_config_mcdc.py ..\ToolConfig\tool_config.py*
*.\runTests.bat ..\TestExecution.txt ..\ToolConfig\tool_config.py*

Of course the test has to be in the list of executed tests (TestExecution.txt).
It consists of the following sub-tests that are reported:

- testDiffOutFiles: checks if the execution is as expected and would detect wrong instrumentation
- testDiffUninstrumentedTrace: checks if the instrumented and uninstrumented code are identical, i.e. would detect "Wrong Behaviour"
- testDiffReportFiles: compare the coverage report with the expected coverage report and would detect errors like "Coverage too high" and "Coverage too low"
- testDiffTERinHTML

### 7.3.7 Test: General/Multicondition

This test ensures that multi-condition coverage is computed correctly. It also checks for the correct behavior. It can be started with the configuration file tool_config_multicondition.py, e.g. by:

*copy /Y ..\ToolConfig\tool_config_multicondition.py ..\ToolConfig\tool_config.py*
*.\runTests.bat ..\TestExecution.txt ..\ToolConfig\tool_config.py*

Of course the test has to be in the list of executed tests (TestExecution.txt).
It consists of the following sub-tests that are reported:

- testDiffOutFiles: checks if the execution is as expected and would detect wrong instrumentation
- testDiffUninstrumentedTrace: checks if the instrumented and uninstrumented code are identical, i.e. would detect “Wrong Behaviour”
- testDiffReportFiles: compare the coverage report with the expected coverage report and would detect errors like “Multicondition Coverage too high” and “Multicondition Coverage too low”
- testDiffTERinHTML

### 7.3.8 Test: StatementCoverage

This test ensures that statement coverage is computed correctly. It also checks for the correct behavior. It can be started with the configuration file tool_config_decision.py, e.g. by:

*copy /Y ..\ToolConfig\tool_config_decision.py ..\ToolConfig\tool_config.py*
*.\runTests.bat ..\TestExecution.txt ..\ToolConfig\tool_config.py*

Of course the test has to be in the list of executed tests (TestExecution.txt).
It consists of the following sub-tests that are reported:

- testDiffOutFiles: checks if the execution is as expected and would detect wrong instrumentation
- testDiffUninstrumentedTrace: checks if the instrumented and uninstrumented code are identical, i.e. would detect “Wrong Behaviour”
- testDiffReportFiles: compare the coverage report with the expected coverage report and would detect errors like “Multicondition Coverage too high” and “Multicondition Coverage too low”
- testDiffTERinHTML

### 7.3.9 Test: MCDC/*

These tests ensure that MC/DC coverage is computed correctly. These also check for the correct behavior. These can be started with the configuration file tool_config_mcdc.py, e.g. by:

*copy /Y ..\ToolConfig\tool_config_mcdc.py ..\ToolConfig\tool_config.py*
*.\runTests.bat ..\TestExecution.txt ..\ToolConfig\tool_config.py*

Of course these tests have to be in the list of executed tests (TestExecution.txt).
These tests consist of the following sub-tests that are reported:

- testDiffExpectedTrace: checks if the execution is as expected and would detect wrong instrumentation
- testDiffUninstrumentedTrace: checks if the instrumented and uninstrumented code are identical, i.e. would detect “Wrong Behaviour”

- testDiffTER: compare the coverage report with the predicted coverage values in the comments of the code and would detect errors like “MC/DC Coverage too high”, “MC/DC Coverage too low” and “Reducing to MC/DC Coverage”
- 
- testDiffTERinHTML

# 8 Licences

The generic TAU is a product of Validas AG and must not be distributed or sublicensed without permission of Validas AG. Validas has granted the license to distribute and sublicense it to Verifysoft. Verifysoft is responsible for the Intra-TAU for Testwell CTC++

It has been developed using Python, ant and Java. The licenses of these components are:

- Python, see http://opensource.org/licenses/Python-2.0
- ANT and ANT Contrib, see http://www.apache.org/licenses/LICENSE-2.0

The TAU has been verified and validated as described in the verification and validation report. For all other requirements it has TCL 1 and might have critical errors that the user has to detect during the review of the results.

VALIDAS AG AND ITS AFFILIATES MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
VALIDAS AG AND ITS AFFILIATES SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF VALIDAS AG AND ITS AFFILIATES HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

# 9 References

[AWL] AnalysisWorkList template located in QKit\TAU_Develop\TAU_Docs\AnalysisWorkList.xlsx

[DO178C] RTCA. DO-178C: Software Considerations in Airbone Systems and Equipment Certification, 1st Edition 2011-12-13.

[DO330] RTCA. DO-330: Software Tool Qualification Considerations 1st Edition 2011-12-13.

[EN50128]: BS EN 50128:2011, Railway applications — Communication, signalling and processing systems — Software for railway control and protection systems, BSI Standards Publication.

[IEC61508] International Electrotechnical Commission, IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems, Edition 2.0, Apr 2010.

[ISO26262] International Organization for Standardization. ISO 26262 Road Vehicles –Functional safety–. 1st Edition, 2011-11-15.

[TAG] Tool Safety Manual for Testwell CTC++

[TCER] Tool Criteria Evaluation Report for Testwell CTC++

[VVRep] Verification and Validation Report for Qkit of Testwell CTC++, see QKit/Documents/QKitVnVReport.pdf

# 10 Appendix: Verification and Validation Report of TAU

Verification of the TAU has been split into verification of generic properties (see Section 10.1) and specific properties of the TAU for Testwell CTC++ (see Section 10.2). For the generic properties there is a generic testsuite contained in the TAU_Develop folder that has been used. For the TAU specific requirements the test suite of the Q-Kit has been reused. Furthermore the TAU has been validated to comply with it's user manual during validation of the Q-Kit (see Section 10.3).

Since this is a test suite that tests the TAU *itself*, the test report messages for successful tests, i.e. test verdict PASS, does not always have to be "Success", there are also tests that provoke "Error" or "Failure" messages.

## 10.1 Generic Requirements for the TAU

The TAU Version 1.15 has been verified from Oscar Slotosch from 8.6.2014 to 9.6.2014. All results have been analyzed using the Analysis work list template [AWL] from and stored in the V&V folder of the QKit. After some improvements mainly in documentation there no findings.

## 10.2 Specific TAU requirements for Testwell CTC++

The following tests have been executed from Olavi Poutanen to validate the correct functions of the TAU on 19.5.2014.

### 10.2.1 Test C99Features

This successfully passed.

### 10.2.2 Test DecisionCoverage

This successfully passed.

### 10.2.3 Test General/Condition

This successfully passed.

### 10.2.4 Test General/Decision

This successfully passed.

### 10.2.5 Test General/Function

This successfully passed.

### 10.2.6 Test General/MCDC

This successfully passed.

### 10.2.7 Test General/Multicondition

This successfully passed.

### 10.2.8 Test MCDC/*

This successfully passed and showed the expected errors and positive results.

### 10.2.9 Test StatementCoverage

This successfully passed.

## 10.3 Validation of the TAU

The validation of the TAU has been executed during V&V of the Q-Kit and documented in the V&V report of the Q-Kit [VVRep].